# CS 421 — Programming Language and Compilers

Welcome!

In today's class:

- **Course intro**
  - **What CS 421 is about**
  - **Class structure**
- **Brief discussion of programming language history**
- **Intro to the OCaml programming language**

# Course staff

Professor: Sam Kamin
kamin@illinois.edu

TAs: Susannah Johnson
sjohnsn2@illinois.edu

Ted Pacyga
tpacyg2@illinois.edu

Rohan Sharma
sharma27@illinois.edu

- **Web: courses.engr.illinois.edu/cs421/sp2013/**
- **Piazza: piazza.com/illinois/spring2013/cs421**

# What you will learn this semeste

- How to implement programming languages
  - Writing lexical analyzers and parsers
  - Translating programs to machine language
  - Implementing run-time systems
- How to write programs in a functional programm language
- How to formally define languages (including the definiti of type rules and of program execution)
- Key differences between statically-typed languages (e.g. Java) and dynamically-typed languages (Python, JavaScr
- Plus a few other things...

# Why learn about lexing and parsing?

Learn basic algorithms for dealing with structu
textual data, applying theory from CS 373:

- **DFAs and regular expressions for "lexing"**
- **Top-down (aka recursive descent) parsing**
- **Bottom-up (aka LR, aka shift/reduce) parsing**

# Why learn about definition and implementation of conventional languages?

- Understand standard languages more precisely
- Complete picture of how programs go from keyboard execution
- Understand translation from high-level language machine language — help understand inefficiencies
- Learn to build language processors (esp. compilers)
- Learn basic concepts for handling structured text d abstract syntax; syntax-directed translation

# Why learn about definition, implementation, and use of functional languages?

- Functional programming is the major alternative progran paradigm to imperative/object-oriented programming.
- Functional programming language features are increasir appearing in mainstream languages
- Functional programming concepts increasingly used in n functional languages, and especially in scripting languag easiest to learn them in a functional programming langu

# How this class operates

- Lectures 9:30–10:45 TR in 1404 SC
- Usually weekly assignments, due Tuesday at 9:30am
- Programming mostly in OCaml, a functional language
- Each class's slides put on web day before class; subse the slides — the ones containing exercises — printed distributed in class
- Two midterms and final (dates t.b.a.)
- Course policies — grade calculations, collaboration/chea policy, lateness — given on web

# Classroom pedagogy

- In-class exercises (ungraded) - These are smaller version what you'll be doing on homeworks and exams.
- Discussion encouraged when doing exercises
- Tablet PCs will be used to help me see what you're doi possibly share with class

# Brief overview of programming languages

- First high-level language (FORTRAN) developed c. 19
  Emphasis was on efficiency.
- As computers got faster, and memories grew, langua
  became less and less about efficiency, more and more ab
  programmer productivity.
- Modern language design involves trade-offs among vari
  issues: efficiency; (short-term) programmer convenier
  (long-term) program maintainability; portability; secur
  parallelizability.

# Capsule history of PLs

| | Conventional | O-o | Scripting | Functional |
|---|---|---|---|---|
| 1957/8 | Fortran | | | LISP |
| 1960's | Algol | Simula 67 | | |
| 1970's | C | Smalltalk | | ML |
| 1980's | | C++, Obj. C | | |
| 1990's | | Java | Perl | Haskell |
| | | | JavaScript | OCaml |
| early 2000's | | C# | Python | |
| | | | Ruby | |

Recent languages: Scala, Clojure, F#, Lua, Go, ...

Others?

# Programming language features

| | Traditional, "static" | Static o-o | Scripting, "dynamic" | Mixed |
|---|---|---|---|---|
| Examples | C, Fortran | C++ | Python, JS | Java, OCaml |
| Objects? | No | Yes | Yes | Yes |
| Automatic mem. mgt.? | No | No | Yes | Yes |
| Static types? | Yes | Yes | No | Yes |
| Tagged values? | No | No | Yes | Yes and no |

# Down to details...

- MP0 — "due" Thursday 9:30am
  - Install OCaml, write simple programs (approx. 1 hr. wo
  - Not graded, but uses handin
- MP1 — due Tuesday, 1/22, 9:30am
  - Recursive functions on lists in OCaml
- First month of class is on lexing, parsing, type-checking, abstract syntax
  - Programming assignments in OCaml
  - Next two classes: intro to OCaml

# OCaml

- **OCaml is a popular functional programming langua which means**
  - Calculations performed mainly by writing recursive functi on lists and trees
  - Automatic memory management
  - No *assignable* variables — variables assigned once, never assigned; no while loops
  - Higher-order functions (but not until later in semester)
- **Also, OCaml makes it very convenient to define manipulate *abstract syntax trees*, which are of cru importance to us in this class.**

# Interactive use of OCaml

Like Python and Ruby, OCaml can be used by entering expressions function definitions interactively, in a "read-eval-print" loop:

```
~$ ocaml
# let pi = 3.14159;;
val pi : float = 3.14159
# let circum r = 2.0 *. pi *. r;;  (* Use *. for floating pt mu
val circum : float -> float = <fun>
# circum 8.0;;
- : float = 50.26544
# let fac x = if x=0 then 1 else x * fac (x-1);;
Error: Unbound value fac
# let rec fac x = if x=0 then 1 else x * fac (x-1);;
val fac : int -> int = <fun>
# fac 4;;
- : int = 24
```

# Interactive use of OCaml (cont.)

```
# let rec binom n m = if m>n then 0
                      else if m=0 or m=n then 1
                      else binom (n-1) m + binom (n-1) (m-1);;
val binom : int -> int -> int = <fun>
# binom 4 2;;
- : int = 6
# #use "defs.ml";;  (* defs.ml contains defn of fib *)
val fib : int -> int = <fun>
# fib 4;;
- : int = 5
```

- Function application syntax: justaposition
- Variable and function names must begin with lower-c letter or underscore; can contain digits or apostrophe.
- No “return” statement: function body is an expression, its value is returned.

# Interactive use of OCaml (cont.)

- `if-then-else` **is a conditional** ***expression*****, not a statem**
  **like C's conditional expression (***condition* ? *expr* : *expr***).**
- `#use` **must be used with care: if you load a file, then us**
  **definitions to define some other functions, and then** ***rel***
  **the original file, the functions you defined in the meant**
  **will still use the old definitions.**
- **Although no types are given, OCaml is a** ***statically ty***
  **language, like C or Java, unlike Python or JavaScript:**

```
# circum 4;;
Error: This expression has type int but an expression was expected of ty
         float
```

# Types in OCaml

- OCaml provides powerful built-in types:
  - Primitive types: int, float, bool, char, string
  - Structured types:
    - Homogeneous lists: $\tau$ list, where $\tau$ is any type
    - Heterogeneous tuples: $\tau_1$ `*` $\tau_2$ `*` $\cdots$ `*` $\tau_n$
    - Functions: $\tau_1$ `->` $\tau_2$ `->` $\cdots$ `->` $\tau_n$
- The class facility allows the definition of new types, a: Java. However, we will not use classes. There is anot way to define new types — using the `type` facility — that will use heavily. We will discuss that next week.

# Primitive types

- Primitive types — int, float, bool, char, string — are mor
  less what you would expect:
  - **int:** `3, 74, -853`
  - **float:** `3.0, .012`
  - **bool:** `true, false`
  - **char:** `'a', '\n'`
  - **string:** `"sam I am"`
- ... with one quirk: arithmetic operations for floats us
  period (+., ∗., etc.)

# Primitive types (cont.)

- **Comparison operations:** =, <, >, <>, <=, >=
- **Boolean operations:** `or` **or** `||`, `&` **or** `&&`, `not`
- **String operations:** ^ **(concatenation); s.[i] (subscript)**
- **No automatic type conversions; use** `string_of_` `int_of_string`, `float_of_int`, **etc.**
- **Various "modules" provide additional operations; th are loaded by entering** `open` ***module-name*** `;;`.
  - **String and Str modules have operations on strings.**
  - **Pervasives is a module that is always loaded — don't need to open it.**
  - **Index of modules is linked at the bottom of the onl manual — see "resources" tab in course web page.**

# Tuples

- Create "struct" by putting expressions in parenthe separated by commas:
  - `(3, 5.0)`: int * float
  - `(3, "abc", true)`: int * string * bool
- Exercise — fill in types:
  - ('a', 'b') :
  - ('a', "a", 'a') :
  - (5, ("a", 'a')) :
- Exercise — create a value of the given type:
  - (3, (4, 5.0), "6") : int * (int * float) * string

# Tuples (cont.)

- Use functions `fst` and `snd` to get elements of a pair. *C*
  *works for pairs.* (We'll see how to deal with bigger tuple
  next class.)

- Exercise: Write a function to add the elements of an int *
  pair:

  - `let addelts p =` fst p + snd p

```
addelts (3,4);;  (* returns 7 *)
```

# Lists

- **Create a list by putting expressions, all of same type, square brackets separated by semicolons:**
  - `[3]`: **int list**
  - `[true; false; true]`: **bool list**
  - `[['a']; ['b'; 'c'; 'd']; []]`: **(char list) list**
  - `[3; 4.0]`: ***type error***
  - `[3; [4]]`: ***type error***

# Lists (cont.)

- **Exercise — fill in types (or flag error):**
  - `['a'; 'b']` : char list
  - `['a'; 'b'; "c"]` : type error
  - `[4; int_of_string "34"]` : int list
  - `[[4]; [5]; []]` : (int list) list
  - `[(1, 2); (3, 4)]` : (int * int) list
  - `[(1, 2); (3, 4, 5)]` : type error
  - `[(1, [3]); (4, [5; 6])]` : (int * (int list)) list

# Lists (cont.)

- **Exercise — create a value of the given type (other than empty list):**
  - [3] : int list
  - [ [3] ] : (int list) list
  - [(3, "3")] : (int * string) list
  - [ ["3"] ] : (string list) list
  - [(3, ["3"])] : (int * string list) list
  - [[(3, ["3"])]] : ((int * string list) list) list

# Lists (cont.)

- **Use @ to concatenate lists, :: to "cons" to start of list; l List module to get functions hd, tl, nth, length, and othe**

```
# open List;;
# let lis1 = [1; 2; 3];;
val lis1 : int list = [1; 2; 3]
# hd lis1;;
- : int = 1
# tl lis1;;
- : int list = [2; 3]
# 4 :: lis1;;    (* N.B. non-destructive cons *)
- : int list = [4; 1; 2; 3]
# [4] @ lis1;;   (* N.B. non-destructive append *)
- : int list = [4; 1; 2; 3]
# length lis1;;
- : int = 3
# nth lis1 2;;
- : int = 3
```

# Lists (cont.)

- **Exercise: Write a function to compute the sum of the f two elements of an int list:** `addfirsttwo [5; 3; 2; 6] =` **You can assume the list is of length at least 2:**
  - `let addfirsttwo lis =` hd lis + hd (tl lis)

- **Exercise: Write a function to compute the sum the *lengths* of the first two elements of an (int l list:** `addfirsttwolengths [[5; 3]; [2]; [6; 2; 5; 3]]` **3. You can assume the list is of length at least 2:**
  - `let addfirsttwolengths lis =`
    length (hd lis) + length (hd (tl lis))

# Polymorphic functions

- OCaml detects the types of functions, so that variable :
function types don't have to be declared.
- But consider:

```
let revpair p = (snd p, fst p)
```

- It is legal to write
  - `revpair (3, 4.0)`
  - `revpair ("abc", true)`
  - *etc.*
- In fact, for any types $\tau$ and $\tau'$,

$$\texttt{revpair:}\ \tau * \tau' \rightarrow \tau' * \tau$$

# Polymorphic functions (cont.)

- **OCaml realizes this and assigns `revpair` a *polymorphic* t**
  **— a type with "type variables" in it:**

```
# let revpair p = (snd p, fst p);;
val revpair : 'a * 'b -> 'b * 'a = <fun>
```

  **Read this as "`revpair` has type $\alpha * \beta \rightarrow \beta * \alpha$."**

- **Similarly, a function may operate on lists of any type. Th**
  **`length` has type $\alpha$ list $\rightarrow$ int.**

# Polymorphic functions (cont.)

Ex: Write the polymorphic types of the following functions. (You can write ei
'a, 'b, etc. or $\alpha$, $\beta$, as you prefer.)

```
let mktriple p = (fst p, snd p, 3)
```

$\alpha * \beta \rightarrow \alpha * \beta * int$

```
let pair_of_first p = (fst p, fst p)
```

$\alpha * \beta \rightarrow \alpha * \alpha$

```
let double_first lis = [hd lis; hd lis]
```

$\alpha\ list \rightarrow \alpha\ list$

```
let pair2list p = [fst p, snd p]
```

$\alpha * \alpha \rightarrow \alpha\ list$

```
addfirsttwolengths (defined above)
```

$\alpha\ list\ list \rightarrow int$

# Wrap-up

- **Today we discussed:**
  - What CS421 is about
  - Basic OCaml programming and use of the top level
- **We discussed it because:**
  - It's good to know why you're learning this stuff
  - We'll be using OCaml for MPs this semester. And one goal of the c
    is for you to learn functional programming.
- **In the next class, we will:**
  - Talk more about OCaml — esp. writing functions on lists.
- **What to do now:**
  - *MP0* — not graded, but using OCaml a little will give you a much be
    feel for it than listening to me talk about it.